Critical Lock Analysis: Diagnosing Critical Section Bottlenecks in Multithreaded Applications

Guancheng Chen[*†] and Per Stenstrom[†]

[*]IBM Research - China, [†]Chalmers University of Technology
chengc@cn.ibm.com, pers@chalmers.se

***Abstract* - Critical sections are well known potential performance bottlenecks in multithreaded applications and identifying the ones that inhibit scalability are important for performance optimizations. While previous approaches use idle time as a key measure, we show such a measure is not reliable. The reason is that idleness does not necessarily mean the critical section is on the critical path.**

We introduce critical lock analysis, a new method for diagnosing critical section bottlenecks in multithreaded applications. Our method firstly identifies the critical sections appearing on the critical path, and then quantifies the impact of such critical sections on the overall performance by using quantitative performance metrics. Case studies show that our method can successfully identify critical sections that are most beneficial for improving overall performance as well as quantify their performance impact on the critical path, which results in a more reliable establishment of the inherent critical section bottlenecks than previous approaches.

***Keywords* – Performance Analysis; Critical Section; Critical Path; Multithreading; Multicore**

I. INTRODUCTION

As the number of cores in multicore systems is increasing, it will be increasingly difficult for multithreaded applications to fully utilize the computing resources. Applications that utilize processing resources (cores and caches) well on a smaller scale can expose new bottlenecks as the number of cores increases. In addition, it will be substantially harder to understand where the bottlenecks are and what their root causes are.

Multithreaded applications built on the shared memory model mainly use locks as the mechanism to coordinate accesses to shared data within critical sections. A well-known performance problem using locks, however, is that lock contention limits the amount of parallelism by serializing accesses to protected shared data. In fact, large and highly contended critical sections can fundamentally limit the potential speedup in future multicore systems [10]. Hence, in order for multithreaded codes to scale to a large number of cores, it is important to identify what critical section bottlenecks will show up if more threads are employed so as to mitigate the bottlenecks through either software changes or architectural approaches such as accelerated critical sections [25], speculative lock reordering [19] or transactional memory [12]. What is needed is a method that allows us to collect information from runs of multithreaded applications that can pinpoint the root cause of critical section bottlenecks and quantify their impact on the overall performance.

Previous research proposed several methods to analyze bottlenecks that are caused by critical sections in multithreaded applications. Approaches taken in these methods encompass quantifying the idleness caused by locks [6, 7, 23, 26], exploiting the potential parallelism of critical sections [18] or characterizing data sharing inside critical sections [20]. While these approaches quantify the execution overhead of critical sections in isolation, they do not consider what impact critical sections have on the completion time of multithreaded applications by taking the critical execution path [14] into account. A finding in this paper is that this may result in misleading conclusions as to what critical sections should be optimized. On the other hand, critical path analysis has been successfully used to identify performance bottlenecks of TCP connections [1], to find program bottlenecks of message passing and shared memory parallel programs [3, 13], to identify code segments (e.g. procedures) that are worthy of optimization [4, 5], to analyze performance bottlenecks across software and hardware [21, 22], and to improve performance of programs [2, 9, 11]. However, these prior efforts do not use critical-path analysis to diagnose critical section bottlenecks quantitatively. The method presented in this paper fills that gap.

In this paper, we introduce *critical lock analysis*, a new method for diagnosing critical section bottlenecks based on quantitative analysis of their impact on multithreaded applications' completion time. The key insight behind our proposed method is that *it is only the critical sections which appear on the critical path that have a direct impact on the completion time of multithreaded applications*. In contrast, other critical sections that are off the critical path may be overlapped by other executions, or they can be executed in parallel. As a result, they do not affect the overall completion time and optimizing these critical sections will not improve performance even if they are contended. Moreover, in order to quantify the criticality of critical section, our method also introduces two quantitative performance metrics – *contention probability* and *size* of a critical section along the critical path. By employing critical lock analysis, we are able to efficiently identify which locks are the most beneficial ones to

SC12, November 10-16, 2012, Salt Lake City, Utah, USA
978-1-4673-0806-9/12/$31.00 ©2012 IEEE

optimize as well as to provide quantitative statistics for guiding the mitigation of critical section bottlenecks.

As a proof of concept, we implement a performance analysis tool based on our critical lock analysis method. Our evaluation, which is driven by a detailed case study of several multithreaded applications, reveals that our method can reliably establish the root causes of critical section bottlenecks in contrast to previous attempts.

In summary, this paper makes the following contributions:

- We introduce critical lock analysis, a method that identifies critical sections appearing on the critical path which are most critical for improving overall performance (identification).
- We introduce two performance metrics: contention probability and size of a critical section along the critical path, which quantify the impact of critical sections on the overall performance of a multithreaded application (quantification).
- We introduce a performance analysis tool employing critical lock analysis with a detailed case study of a set of multithreaded applications, which demonstrates the effectiveness and in fact the necessity of critical lock analysis (proof of concept).

The rest of the paper is organized as follows. In Section II, we introduce the basic concepts on which our method is built by way of an illustrative example. The algorithm and implementation of critical lock analysis are described in Sections III and IV, respectively. The case study is presented in Section V. We contrast our proposed method and our findings with the known body of work in Section VI before concluding in Section VII.

II. Basic Concepts

An execution of a multithreaded program using four threads is shown in Fig. 1. We distinguish program sections that are not protected by locks (non critical sections) from sections that are protected (critical sections). Critical sections are guarded by four locks: L1, L2, L3 and L4. It is clear that the completion time of the example program is directly affected by the length of the critical path which is illustrated in Fig. 1 by a line through the longest set of sections.

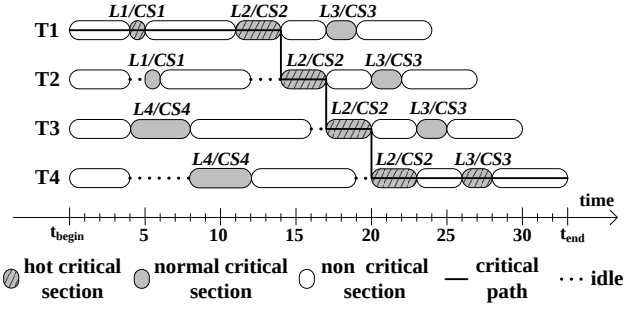

Figure 1. An execution of a multithreaded application and its critical path. *Hot critical sections* are guarded by *critical locks*; *normal critical sections* are guarded by *normal locks*.

The (six) critical sections included on the critical path are called *hot critical sections* and the locks protecting the hot critical sections are called *critical locks*. Analogously, we refer to the other critical sections/locks as *normal critical sections/locks*. The difference between a critical and a normal lock is that the former affects the overall performance directly and thus should be the first target for optimization. In other words, optimizing the critical locks which appear on the critical path provides us with a better opportunity for improving overall performance.

If we optimize CS1 guarded by L1 with a small value ε, it is expected that the total completion time for this execution will be reduced by ε because T1's invocation of CS1 appears once on the critical path. On the other hand, if we optimize CS2 protected by critical lock L2 with a small value ε it is expected that the execution time will be shortened by 4ε, suggesting that L2 has a bigger performance impact than L1. By contrast, even though CS4, that is invoked by T3, introduces lock contention for T4, optimizing CS4 will not have any impact on the completion time since its execution is overlapped by the critical path.

Previous critical section bottleneck analysis methods [6, 7, 18, 20, 23, 26] mainly focus on quantifying the idleness caused by lock contention and do not take the critical path into account. For example, these methods would indicate L4 in Fig.1 as the most important critical section bottleneck since it introduces the longest idle time. As a result, efforts could be wasted on optimizing this normal lock which does not have a direct impact on the overall performance.

It is worth noting that a critical lock may not necessarily introduce idleness. For example, a hot critical section, such as T4's invocation of CS3, which is protected by an uncontended critical lock L3, also contributes to the completion time. Indeed, by quantifying the criticality of locks on the critical path, rather than the idleness caused by locks, our method can reveal a deeper insight of critical section bottlenecks than previous approaches as shown in Fig.1.

Owing to the fact that a single lock can be used to protect several different critical sections, performance metrics of critical sections should be aggregated and attributed to their shared lock together if that is the case. As a result, the method described in the next section measures the performance impact of locks instead of critical sections.

III. Critical Lock Analysis

Our critical lock analysis is based on *critical path analysis* [14], a general method which has been successfully used to identify other performance bottlenecks such as critical code segments (e.g. procedures) [4, 5, 13] and critical TCP connections [1]. For a multithreaded application, critical section bottlenecks can delay the completion of threads and thus result in poor performance. By conducting critical lock analysis, it is possible to identify the critical locks on the critical path and quantify the impact of critical locks on the overall performance of the multithreaded application.

This section first describes an algorithm for identifying the critical locks appearing on the critical path in Section III.A. We then discuss two proposed quantitative performance metrics of the algorithm in Section III.B.

A. Algorithm

To describe the algorithm for identifying the critical locks appearing on the critical path, we define a *segment* as the executed code of a thread between two synchronization events which might introduce blocking. Typical synchronization events in multithreaded applications are caused by *locks*, *barriers* and *condition variables* as well as *thread creation* and *thread termination*. As a result, locks comprise the start or end point of some segments since not all segments contain lock operations. The algorithm assumes that we have collected run-time information about *when* and *by which* thread the synchronization events are called and blocked. Pseudo-code for the algorithm is shown in Fig. 2.

```
1  seg = find_the_last_segment();
2  stop = find_the_first_segment();
3  while (seg != stop) {
4      if (segment_contains_lock(seg)) {
5          lock = find_the_lock_of_this_segment(seg);
6          mark_as_critical_lock(lock);
7      }
8      if (segment_blocked_in_the_beginning(seg)) {
9          /* need to change to another thread now */
10         seg = find_the_segment_released_me(seg);
11     }
12     else {
13         /* stay at the current thread */
14         seg = find_the_previous_segment(seg);
15     }
16 }
```

Figure 2. Pseudo-code of critical lock analysis algorithm.

To show how the algorithm conceptually works we use the example program of Fig. 1 again. The algorithm starts from the last segment of the last finished thread T4. It continues backwards until reaching segment CS2 of T4 which starts with a blocking event due to contention on L2. The algorithm establishes that segment CS2 of another thread T3 released the lock L2 that CS2 of T4 was blocked on. The algorithm then continues backwards with T3 until it was also blocked. Thus, the algorithm will continue in this manner backwards until the beginning of the execution. All locks that the algorithm goes through during the progress are the critical locks.

B. Two Quantitative Performance Metrics

As pointed out by Eyerman et al. [10], the potential speedup of a multithreaded application is not only limited by the sequential part, but is also fundamentally limited by the contention probability and the size of critical sections. However, their model simply assumes that all critical sections are equally critical to overall performance. We extend their work to critical lock analysis in a fundamental way by introducing two performance metrics to quantify the impact of critical locks on the overall performance in the following subsections.

1) Contention Probability of Critical Lock

We define the *contention probability of a critical lock* as

$$P = \frac{number\ of\ contended\ invocations\ of\ a\ critical\ lock}{number\ of\ all\ invocations\ of\ a\ critical\ lock}$$

Consider for example the critical lock L2 in Fig. 1. Its contention probability is 75% (3/4) on the critical path. Correspondingly, L2 is invoked 4 times on the critical path, which is an increase of four times compared to the average invocation number of one time for each thread. When a critical lock introduces high contention, such as L2, it will *likely* block many other threads and, thus, will result in a dramatic increase of the appearance frequency on the critical path as suggested by the way the algorithm works. A counter-example is the critical lock L1 invoked by T1. Its contention probability on the critical path is 0 hence its impact on the overall performance is limited to the size of its single invocation on the critical path.

2) Size of Hot Critical Section

We define the *size of a hot critical section* as the time spent on executing the hot critical section. For example, in Fig. 1, three time units are spent on executing each of the four invocations of the hot critical section CS2, which in total accounts for 36.36% (4*3/33) of the critical path. On the other hand, CS1's execution only cost one time unit of T1 and accounts for 3.03% (1/33) of the critical path, which is much smaller.

3) Discussion

The implication behind the critical lock analysis algorithm is that the higher contention a critical lock introduces, the more time it will spend on the critical path. At the same time, the larger a hot critical section is, the bigger fraction it will account for on the critical path. If a critical lock is highly contended along the critical path as well as has a large hot critical section size, this critical lock is likely to dominate the critical path of the multithreaded application. Thus, it will have a big impact on the overall performance.

Previous methods [6, 7, 18, 20, 23, 26] mainly use the metric of *idleness* caused by lock contention to analyze critical section bottlenecks. While quantifying idleness is generally useful, it fails to establish the impact of critical section bottlenecks on the overall performance. For example, even though the contended lock L4 introduces the longest idle time in Fig.1, the idleness caused by this lock does not affect the overall performance directly and optimizing it will not improve overall performance. On the other hand, some uncontended locks, such as L3 invoked by T4, also contribute to the overall performance. As a result, only quantifying the idleness caused by contended locks is not accurate enough to diagnose critical section bottlenecks successfully. In contrast, the two performance metrics we propose do not only quantify the impact of critical section bottlenecks on the overall performance, they also provide a deep insight into the root causes of critical section bottlenecks.

The input of the algorithm needs to be collected by conducting instrumentation and run-time tracing of the target

application. It is important that typical blocking events such as contention for a lock, waiting for a barrier, waiting for a condition variable etc. must be monitored properly for this algorithm to work. How this is done is explained in the next section.

IV. Implementation Of Algorithm

As a proof of concept, we develop a performance analysis tool relying on the described critical lock analysis method. The implementation environment uses Linux as the operating system and POSIX threads (Pthreads[1]) as the threading library.

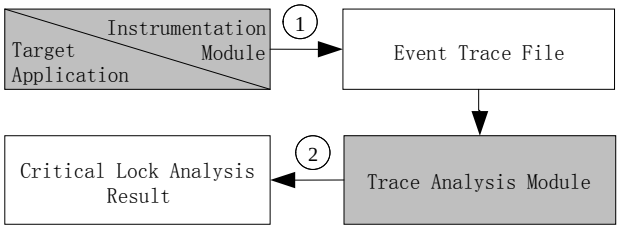

Figure 3. Workflow of our implemented tool. The instrumentation module generates a trace that consists of the target application's run-time behaviour. The analysis module then conducts critical lock analysis based on the trace file.

As shown in Fig. 3, the workflow of our tool consists of two major steps. Firstly, the instrumentation module inserts extra code at each synchronization point in the target application. When this code is executed, it collects corresponding event records, which are stored in memory before they are flushed onto disk when the instrumented target application completes. Secondly, the post-processing analysis module conducts critical lock analysis on the traced data according to the algorithm described in Section III and provides comprehensive performance statistics in the end.

A. Instrumentation Module

In order to capture the needed run-time information for the critical lock analysis algorithm, we instrument all the synchronization routines of Pthreads, which can block a thread. The instrumentation module is implemented as a dynamically linked library, which is preloaded at the launch time of the target application in order to override the original Pthreads routines such as `pthread_mutex_{lock,unlock}`, `pthread_barrier_wait`, `pthread_cond_wait`, `pthread_create`, `pthread_exit`, `pthread_join` etc. As a result, every function call to the Pthreads library from the monitored application is redirected to our manipulated routines in which extra code is executed before or after calling the real Pthreads routines.

The key challenge for implementing the instrumentation module is that we need to not only keep the instrumentation overhead at a low level, but also to collect necessary run-time behavior needed by the critical lock analysis such as when a synchronization event happened. To minimize the instrumentation overhead, we leverage the `mftb`[2] instruction provided by our POWER7-based testing system to read the time stamp counter from the user space in a lightweight manner. As a user space instruction, `mftb` allows us to collect timestamps of synchronization events with minimal intrusion.

```
int pthread_mutex_lock(&mutex) {
    MAGIC();  /* acquire the lock */
    int r = real_pthread_mutex_trylock(&mutex);
    if (r == EBUSY) {
        MAGIC();  /* lock contention */
        r = real_pthread_mutex_lock(&mutex);
    }
    MAGIC();  /* obtain the lock */
    return r;
}

int pthread_mutex_unlock(&mutex) {
    int r = real_pthread_mutex_unlock(&mutex);
    MAGIC();  /* release the lock */
    return r;
}

int pthread_barrier_wait(&barrier) {
    MAGIC();  /* reach the barrier */
    int r = real_pthread_barrier_wait(&barrier);
    return r;
}

int pthread_cond_wait(&cv, &mutex) {
    int r = real_pthread_cond_wait(&cv, &mutex);
    MAGIC();  /* woken up by signal */
    return r;
}

int pthread_cond_signal(&cv) {
    int r = real_pthread_cond_signal(&cv);
    MAGIC();/* signal sent already */
    return r;
}
```

Figure 4. Implementation sketch for the instrumentation module.

The implementation sketch for the instrumentation module is shown in Fig. 4. When the instrumented `MAGIC()` routine is invoked, it will record information including timestamp of synchronization event, event type, identifier of synchronization object and identifier of thread. These records are stored in variables associated with the thread that executed it. When the instrumented application completes, these records will be flushed onto disk as a trace file, which will be fed into the analysis module for further processing.

Thanks to the fact that the instrumented `MAGIC()` code uses the lightweight `mftb` instruction, the instrumentation overhead is very small (around 5% in the 24-thread scenario of the applications we use) and hence we are able to capture actual run-time behavior as well as to scale our method to a large number of threads.

The specific instrumentation strategy for different kinds of synchronization primitives is described below.

1) Lock

[1] Although we use Pthreads in our implementation, our method is also applicable to other threading libraries which use locks such as OpenMP since the principle is the same.

[2] `rdtsc` is the equivalent of `mftb` on x86 architecture.

A particular lock operation can be divided into three steps: acquire the lock, obtain the lock and release the lock. Lock contention could happen between acquiring and obtaining the lock. As a result we instrument with `MAGIC()` in the four corresponding positions around the `real_pthread_mutex_{lock, unlock}` routines as shown in Fig. 4. The time spent by a thread between obtaining and releasing a lock is the *hold time* for the lock. For a contended lock request, the time spent between acquiring and obtaining a lock is the *wait time* for the lock.

We firstly try to acquire the lock by calling the `trylock` routine. If the request is not blocked, `trylock` will work in the same way as a `lock` operation, otherwise it will switch to a block waiting `lock` until obtaining the lock. Moreover, we choose to instrument with `MAGIC()` after the `unlock` operation because it will not introduce extra overhead inside the critical section.

2) Barrier

For a barrier synchronization event, all previously arrived threads will be blocked until the last thread reaches the barrier. As a result, the `MAGIC()` for `barrier_wait` is inserted just before the routine so that we can record the arrival time of each thread.

3) Condition Variable[3]

When a thread calls `cond_wait`, it will be blocked until receiving the desired signal from another thread calling `cond_signal`. By inserting `MAGIC()` after `cond_wait` and `cond_signal`, we will be able to capture which thread blocked the thread waiting for a condition variable.

B. Analysis Module

The post-processing analysis module will conduct critical lock analysis based on the collected traces of the target application. A key point of the implementation is to identify the correct thread which released a blocked synchronization primitive. For locks, the thread holding the same lock adjacently before the blocked thread is the desired one. For barriers, the thread reaching the same barrier lastly is the desired one. For condition variables, the thread signaling the same condition variable to the blocked thread is the desired one.

V. CASE STUDY

To show the benefits derived from critical lock analysis, one micro-benchmark and several real applications are used. Section V.A describes our experimental environment, Section V.B presents the results of the micro-benchmark and Section V.C, V.D and V.E show the results of the real multithreaded applications, respectively.

[3] Although it is possible, we will not address the impact of barrier and condition variables on the overall performance in this paper since we mainly focus on diagnosing critical section bottlenecks.

A. Experimental Environment

The configurations of the machine and the three real applications can be found in Table 1. The machine is a 24-thread POWER7 system (2 processors * 6 cores * 2-way SMT) running Linux 2.6.32. We use a micro-benchmark that we discuss in the next subsection to highlight the advantage of critical-lock analysis compared to other methods. In addition, we also use a set of real applications: Radiosity, Water-nsquared, Volrend and Raytrace from SPLASH-2 [28], a Pthreads implementation of the Travelling Salesman Problem (TSP) [27], Unbalanced Tree Search (UTS) benchmark [16] and OpenLDAP 2.4.21 [17]. All these benchmarks use multiple locks frequently and thus are good candidates for examining the effectiveness of our method to a broad set of multithreaded applications.

Table 1. Configuration of test machine and applications

Core Number	12
Hardware Thread Number	24 (SMT2 model)
Architecture	POWER7
Memory	64GB
OS	Linux 2.6.32
Radiosity input	-batch -largeroom
Water-nsquared input	512 molec
Volrend input	head
Raytrace input	car 256
TSP input	10 cities
UTS input	-T 8 -c 2 $T3
OpenLDAP 2.4.21 input	10k directory entries

B. Micro-benchmark Result

Obviously, while a micro-benchmark does not give an accurate portrayal of critical section bottlenecks in real applications, it is useful for gaining insight into the advantages provided by our method compared to previously proposed approaches.

```
pthread_mutex_lock(&L1);
for (i=0; i < 2000000000; i++) {
    a++;
}
pthread_mutex_unlock(&L1);

pthread_mutex_lock(&L2);
for (j=0; j < 2500000000; j++) {
    b++;
}
pthread_mutex_unlock(&L2);
```

Figure 5. Pseudo-code for the micro-benchmark.

As shown in Fig. 5, the micro-benchmark we use consists of two consecutive locks (L1 and L2 in Fig. 5) protecting two different counters, which reside in different cache lines to eliminate false sharing. In each critical section, the protected counter is incremented by all threads in a loop, one at

a time, and the loop is iterated 2 billion times and 2.5 billion times in the first and second critical section, respectively.

In order to help examine the effectiveness of our method, we provide two types of statistics, which contrast our method to previous approaches as shown in Table 2. The first type of statistics (*TYPE 1*) is based on our critical lock analysis, which includes information for *each critical lock* such as the fraction of critical path time that a hot critical section accounts for, and the contention probability of a critical lock (along the critical path). The second type of statistics (*TYPE 2*) is provided for *each lock* (which can either be a critical lock or a normal lock) in the same way as used by previous approaches [6, 7, 18, 20, 23, 26] which do not take the critical path into account. More specifically, it provides information such as wait time, hold time, and contention probability of each lock.

Table 2. TYPE 1 and TYPE 2 represent two different types of statistics provided by our critical lock analysis and previous approaches, respectively.

TYPE 1	***Description***
CP Time %	The fraction of the critical path time that the hot critical sections protected by a critical lock accounts for
Invocation # on CP	The number of invocations of a critical lock along the critical path
Cont. Prob. on CP %	The contention probability of the invocations of a critical lock along the critical path

TYPE 2	***Description***
Wait Time %	The average fraction of time each thread spent on waiting for the lock
Avg. Invo. #	The average number of invocations of a lock by each thread
Avg. Cont. Prob %	The average contention probability of a lock by each thread
Avg. Hold Time %	The average fraction of time each thread spent on executing the critical sections protected by a lock

The first goal of our method is to identify the locks that are most beneficial for improving overall performance. In Fig. 6 we show the results in a 4-thread scenario. The second and third columns represent the results provided by our method and previous methods [6, 7, 18, 20, 23, 26], respectively: *CP Time* means the fraction of the critical path time that the hot critical sections protected by a critical lock accounts for; *Wait Time* means the average fraction of time each thread spent on waiting for the lock. In other words, CP Time belongs to TYPE 1 and represents the total hold time of a critical lock on the critical path, whereas Wait Time belongs to TYPE 2 and represents the average wait time of all the invocations of a lock (including both critical lock and normal lock).

Lock	***CP Time %***	***Wait Time %***	***Speedup after optimization***
L1	16.67%	***36.33%***	1.26
L2	***83.33%***	9.02%	***1.37***

Figure 6. Results of the two locks of the micro-benchmark employing 4 threads. The second column belongs to TYPE 1 statistics and the third column belongs to TYPE 2 statistics.

The bold italic numbers in Fig. 6 indicate the most beneficial lock for optimization as suggested by the two corresponding methods. Our method indicates that L2 is the most important bottleneck according to the CP Time metric, whereas previous methods suggest that L1 is the most beneficial target for optimization since it introduces the highest Wait Time.

To validate which lock is the most important bottleneck in practice, the speedups after optimizing the two locks respectively are measured. We optimize the two critical sections by reducing the iteration count of the two protected loops by 1 billion times, respectively, which accounts for the same amount of optimization efforts. As we can see in Fig. 6, optimizing L2 provides better speedup than L1 using the same optimization effort. This shows that previous approaches that focus on quantifying idleness caused by lock contentions may lead to wrong conclusions regarding critical section bottlenecks and makes a strong argument that critical lock analysis is needed for a reliable assessment.

To better explain the reason why L2 is more important for overall performance, we illustrate a representative execution of the micro-benchmark in Fig. 7. As we can see, even though L1 introduces more idle time, its execution is overlapped by the critical path, which is dominated by CS2 protected by L2. As a result, L2 is the most beneficial optimization target for improving overall performance in practice.

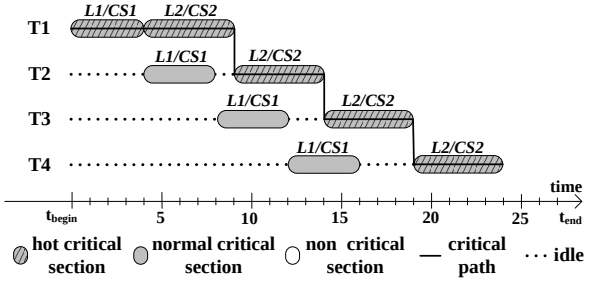

Figure 7. A representative execution of the micro-benchmark.

C. Real Application Results

To show that our method is applicable to a broad set of multithreaded applications, we compare the critical lock analysis result with previous approaches for several benchmarks. Fig. 8 shows the impact of CP Time and Wait Time on the critical path time for the two most important locks in these applications.

As we explain in Section II and demonstrate in the micro-benchmark results, critical lock analysis is fundamentally more reliably on diagnosing critical section bottlenecks by quantifying the end-to-end impact of locks on the overall performance, in contrast to considering idleness alone. For

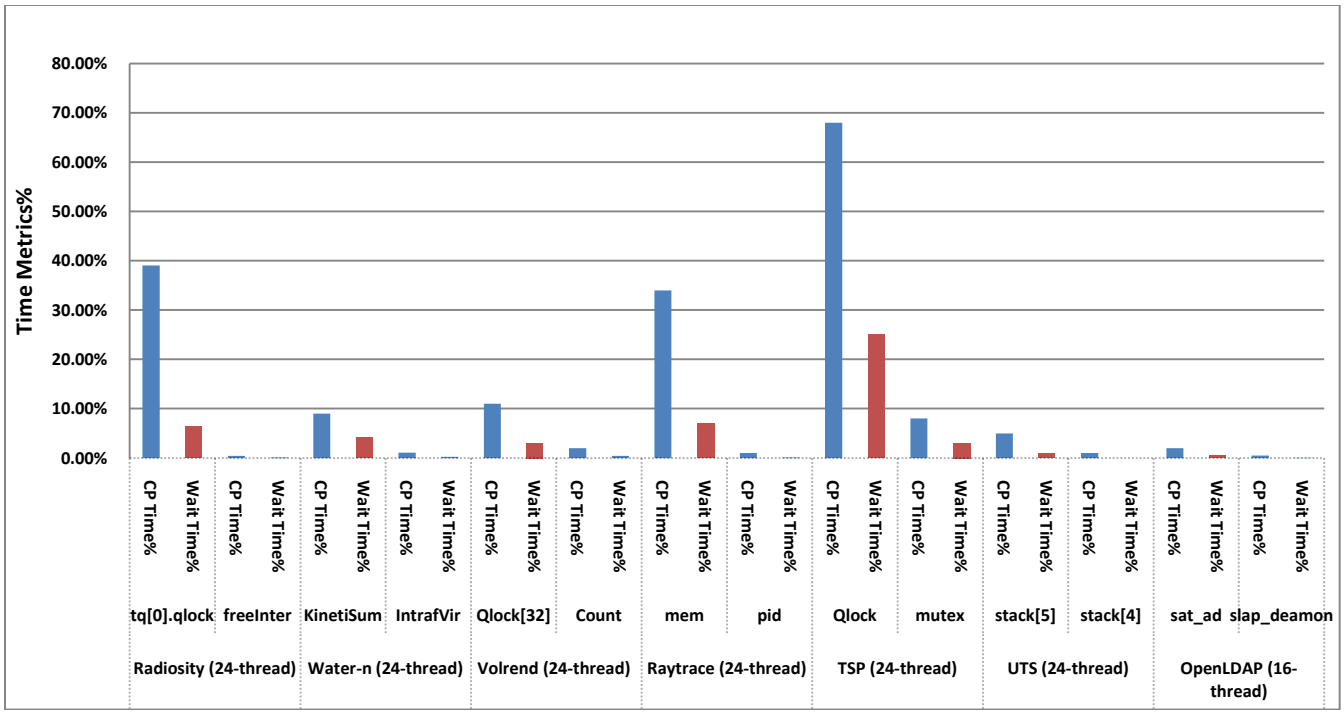

Figure 8. Results of the two most critical locks for a set of multithreaded applications. "CP Time%" column belongs to TYPE 1 statistics and the "Wait Time%" column belongs to TYPE 2 statistics.

example, the Wait Time metric significantly underestimates the performance impact of `tq[0].qlock` of Radiosity, `mem` of Raytrace and `Qlock` of TSP compared to the CP Time metric. As a result, a programmer can be misled and miss the optimization opportunities of these locks if he/she considers idleness of critical section bottlenecks as the main metric. This provides a strong evidence that quantifying idleness alone is not enough to evaluate the impact of critical section bottlenecks on the overall performance.

For applications where critical sections are not an important bottleneck, our method can still help. As we can see from Fig. 8, the CP Time metric indicates that `stackLock[5]` of UTS accounts for 5% of the critical path, whereas the Wait Time metric suggests that this lock is not a performance bottleneck as it indicates that `stackLock[5]` does not introduce significant lock contention. However, `stackLock[5]` still appears on the critical path and thus affects the overall performance directly. It is clear that our method can provide a more reliable insight when diagnosing critical section bottlenecks in such a scenario than considering idleness alone.

We also examine complex real-world applications to examine the usefulness of our tool. OpenLDAP is an open source implementation of the Lightweight Directory Access Protocol and is used in a variety of data communication applications. This application is used to show that our method is applicable for diagnosing critical section bottlenecks for non-trivial real-world applications. To examine the critical section bottlenecks of OpenLDAP, we bind SLAMD [24] on a dedicated core on the same machine to generate 10k search requests for the OpenLDAP server application running with 16 threads on other cores. As we expect, our tool reveals that critical sections are not a significant bottleneck for OpenLDAP as shown in Fig. 8. After examining the source code, we find that the locks are either tuned in a fine-grain manner or are only invoked for limited times which results in negligible critical section bottlenecks. Unsurprisingly, since OpenLDAP has been improved for more than 10 years, critical section bottlenecks have been eliminated in this process.

After identifying the most critical locks for a multithreaded application, our method can quantify the impact of the critical locks on performance by using the two proposed quantitative metrics. In the following subsections, we demonstrate how root causes of critical section bottlenecks are revealed and how this can guide optimization efforts in especially Radiosity and TSP, which have the two highest CP Time among the applications we examine.

D. Radiosity Results

Radiosity is an implementation of a global illumination algorithm used in computer graphics. Radiosity is interesting because the scalability of it is relatively poorer than other applications in SPLASH-2 and it uses locks extensively. As a result, we suspect its performance is limited by critical section bottlenecks. Note that this application has been carefully tuned and designed to eliminate false sharing. To examine the usefulness of our method, we profile the parallel phase of Radiostiy running with 4, 8, 16, 24 threads.

Lock	4 threads		8 threads		16 threads		24 threads	
	CP Time %	Wait Time %	CP Time %	Wait Time %	CP Time %	Wait Time %	CP Time %	Wait Time %
tq[0].qlock	***0.53%***	***0.20%***	0.30%	***0.12%***	***5.40%***	***2.00%***	***39.15%***	***6.40%***
freeInter	0.44%	0.01%	***0.51%***	0.02%	0.53%	0.06%	0.45%	0.08%

Figure 9. Results of the two most critical locks of Radiosity.

1) Identification

Fig. 9 illustrates the impact on the critical path of the two most important locks (sorted by CP Time) as determined by the critical lock analysis method. It is clear that in the 8-thread configuration, `freeInter` is the most important lock according to the CP Time metric, but only the second most important lock based on Wait Time. Again, this shows that only quantifying idleness is not accurate enough for identifying the most important critical section bottlenecks and our method is critical in this role.

The results in Fig. 9 also reveal that `tq[0].qlock` starts to dominate the critical path when the number of threads increases. It is interesting to see that when 24 threads are employed, the hot critical sections protected by this lock occupy around 39% of the critical path. This indicates that `tq[0].qlock` is the most critical lock when more than 8 threads are used.

It is also clear that the difference between the weights assigned to each lock by the Wait Time metric and CP Time metric is pronounced. For example, although both metrics indicate `tq[0].qlock` as the most critical lock for tuning in the 24-thread configuration, CP Time assigns a significantly higher weight to this lock (39.15%) than Wait Time does (6.40%).

2) Quantification

To better understand why `tq[0].qlock` becomes the most critical lock, we use the proposed two quantitative metrics to help reveal the root causes. As we will show in the following subsections, the reason that the critical path fraction of `tq[0].qlock` is high when 24 threads are used comes from two factors: high contention probability of the critical lock and the size of the hot critical section.

a) Contention Probability

Lock	Invo. # on CP	Cont. Prob. on CP %	Avg. Invo. #	Avg. Cont. Prob %	Incr. Times of Invo. #
tq[0].qlock	26298	78.69%	3751	70.16%	7.01
freeInter	13127	9.31%	9182	0.13%	1.43
tq[18].qlock	378	35.17%	193	11.21%	1.95

Figure 10. Contention probability statistics of Radiosity employing 24 threads (Partial). The second and third columns belong to TYPE 1; the fourth and fifth columns belong to TYPE 2.

Fig. 10 presents the contention probabilities of the three most critical locks (sorted by CP Time) under the 24-thread configuration in our system. It shows that 78.69% of the invocations of `tq[0].qlock` along the critical path are contended. High contention probability means this critical lock introduces many blockings for other threads along the critical path, and thus it is included many more times in the critical path compared to its average number of invocations by each thread. Correspondingly, 26298 invocations of `tq[0].qlock` are included in the critical path, which is an increase of 7.01 times compared to 3751 average invocations by each thread. By contrast, only 9.31% of the invocations of `freeInter` on the critical path are contended and thus the number of its invocations on the critical path is only slightly more than the average number of invocations by each thread; 13127 compared to 9182 (an increase of 1.43 times).

b) Critical Section Size

In Fig. 11 we present the critical section sizes of the three most critical locks (sorted by CP Time) under the 24-thread configuration. We can see from Fig. 11 that, on average, each thread spends 4.76% of its time on executing the different critical sections protected by `tq[0].qlock` (when 24 threads are used). The large amount of time spent on critical sections contributes to the high critical path fraction of this lock (39.15%). This indicates that its big critical section size together with its high contention probability contribute to its big fraction on the critical path. In contrast, the critical section size of `tq[18].qlock` is relatively small (0.03% for each thread), hence its fraction of the critical path is negligible, even if its contention probability along the critical path is high (35.17% as shown in Fig. 10).

Lock	CP Time %	Avg. Hold Time %	Incr. Times of Critical Section Size
tq[0].qlock	39.15%	4.76%	8.22
freeInter	0.54%	0.43%	1.26
tq[18].qlock	0.28%	0.03%	9.33

Figure 11. Critical section size statistics of Radiosity employing 24 threads (Partial). The second column belongs to TYPE 1; the third column belongs to TYPE 2.

3) Validation

In this section, the objective is to validate that by tuning the critical lock(s) that has been identified as the most significant critical section bottleneck in the critical lock analysis, we can improve the performance of the a multithreaded application.

Radiosity uses 14 locks to protect different shared data structures. Our method identifies that `tq[0].qlock` is the

most critical lock when more than 8 threads are used. Thus we would expect a performance boost when the cost associated with this lock is mitigated. In order to verify this we optimize the critical section protected by lock `tq[0].qlock`.

After analyzing the source code of Radiosity we find that `tq[0].qlock` is used to protect one of the shared task queues. In Radiosity, each thread manages a task queue with tasks that could be dequeued either by the thread itself or by another thread. Moreover, both the `enqueue` and `dequeue` operations need to hold the lock which limits the scalability of Radiosity significantly.

However, quite often, it happens that one thread tries to `enqueue` a task at the tail while another thread that steals a task tries to `dequeue` it from the head. Since most of the operations are independent, we could use a finer-grain lock mechanism to achieve a better performance. We optimize this critical lock by implementing a fine-grain two-lock concurrent queue algorithm as described in [15].

In the optimized version of Radiosity, we replace the `q_lock` with two separate locks: `q_tail_lock` and `q_head_lock`. As a result, the `enqueue` operation only needs to grab the tail lock, whereas the `dequeue` operation only needs to grab the head lock. After the optimization, we get a performance improvement as high as 7%[4] when 24 threads are used as shown in Fig. 12. It is worth noting that after the optimization, other segments which are off the critical path before the optimization can end up on the critical path. Thus the optimization results in 7% performance improvement of end-to-end execution time (i.e. the length of the critical path is reduced by 7%), which is lower than the CP Time of `tq[0].qlock` (i.e. 39.15%).

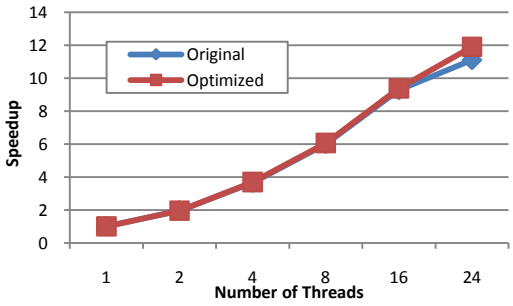

Figure 12. Speedups of original and optimized Radiosity.

To explain where the speedup comes from, the detailed statistics for the optimized Radiosity is shown in Fig. 13 and Fig. 14. After the optimization, `tq[0].q_head_lock` becomes the most critical lock and it corresponds to 2.53% of the critical path, which is much less than the faction of `tq[0].q_lock` before.

Lock	*CP Time %*	*Avg. Hold Time %*	*Incr. Times of Critical Section Size*
tq[0].q_head_lock	2.53%	0.73%	3.47
freeInter	1.12%	0.85%	1.31
pbar_lock	0.18%	0.10%	1.80

Figure 13. Critical section size statistics of an optimized version of Radiosity employing 24 threads (Partial).

As we can see, the dramatic decrease of contention probability and critical section size of `tq[0].q_head_lock` is the root cause of the better performance. More specifically, the average hold time of this lock reduces to 0.73% for each thread, and the contention probability of this lock on the critical path reduced to 53.62%.

We also optimize the other critical locks, which were suggested with a negligible performance impact, such as `freeInter`. Not surprisingly, optimizing these critical locks has little impact on the overall performance. This confirms that the critical locks which occupy high fractions of the critical path are the most important critical section bottlenecks and the most beneficial targets for optimization.

Lock	*Invo. # on CP*	*Cont. Prob. on CP %*	*Avg. Invo. #*	*Avg. Cont. Prob %*	*Incr. Times of Invo. #*
tq[0].q_head_lock	2981	53.62%	892	48.81%	3.34
freeInter	14523	21.42%	9083	14.13%	1.59
pbar_lock	1829	23.18%	876	24.27%	2.08

Figure 14. Contention probability statistics of optimized Radiosity employing 24 threads (Partial).

In summary, optimizing the critical locks pointed out by our method is certainly beneficial for improving the overall performance. This gives us confidence in the correctness and effectiveness of our method.

E. TSP Result

The last case study is based on a Pthreads version of the Travelling Salesman Problem (TSP) benchmark. A global task queue protected by `Qlock` is used by TSP to maintain the paths which is accessed by all threads from time to time. As we have shown in Fig. 8, `Qlock` contributes to 68% of the critical path and hence slow down the overall performance significantly.

To optimize `Qlock`, we use a similar strategy as Radiosity. That is, by splitting `Qlock` into `Q_headlock` and `Q_taillock`, we can parallelize the enqueue and dequeue operations for the global queue, and thus improve the performance of TSP by 19% when 24 threads are used.

VI. RELATED WORK

A. Critical Section Bottleneck Analysis

Independent of our research, several methodologies have been proposed for analyzing critical section bottlenecks. Tal-

[4] It is worth noting that it is possible to achieve even better performance by optimizing tq[0].qlock with more advanced methods such as [12, 19, 25]. Since this is not the main concern of our work, we will not address this issue in this paper.

lent et al. used a strategy to attribute the idleness of locks to the lock holders [26]. Tools such as PinCS predict how much disjoint access parallelism of a critical section can be exploited [18]. There are also tools which can analyze lock contention by providing timing and contention related properties (typically idleness metric) of a critical section [6, 7, 23]. Sahelices et al. propose a method for characterizing data sharing inside critical sections [20].

However, these attempts do not take critical paths into account. As we have shown, this is important because only the critical sections appearing on the critical path contribute to the overall performance directly. As a result, one can efficiently utilize the optimizing efforts by focusing on the critical locks rather than wasting time on optimizing normal locks. In addition, we also show that in order to reveal the root causes of critical section bottlenecks, a successful method should at least provide two more performance metrics – contention probability of critical locks and size of hot critical sections. This is because these two metrics do not only quantify the impact of critical sections on the overall performance, but also are the factors that fundamentally limit the speedup of multithreaded application [10]. To the best of our knowledge, our critical lock analysis is the first method that (i) identifies which critical sections are most beneficial for improving overall performance and (ii) quantifies the impact of critical sections on the overall performance.

B. Critical Path Analysis

Critical path analysis has been used successfully for identifying other performance bottlenecks. Barford et al. proposed a critical path analysis based method for analyzing the performance of TCP connections [1]. Hollingsworth and Böhme et al. designed a run-time critical path profiling algorithm which can be used to find program bottlenecks of message passing and shared memory parallel programs [3, 13]. Saidi et al. used critical path analysis to identify performance bottlenecks across software and hardware [21, 22]. Intel Thread Profiler [4] and Broberg et al. [5] conducted critical path analysis to identify the code segments (e.g. procedures) on the critical path which have the biggest impact on the overall performance. However, neither [4] nor [5] targeted quantitative analysis of critical section bottlenecks by using critical path analysis with quantitative performance metrics. In short, to the best of our knowledge, our work is the first that applies critical path analysis to diagnosing critical section bottlenecks quantitatively.

Critical path analysis has also been used for guiding performance optimization in many cases. Fields et al. suggested a hardware predictor of instruction criticality and used it to improve performance [11]. Bhattacharjee et al. proposed a hardware predictor of thread criticality which can help improve both performance and energy [2]. Dooley et al. employed a critical path analysis based method for online performance tuning of message driven parallel programs [9]. These optimization works suggest a valuable direction of our future work.

VII. Conclusion and Future Work

In this paper we introduce critical lock analysis as a method to identify critical sections in multithreaded applications that appear on the critical execution path. Focusing on optimizing these critical locks offers us a better opportunity for performance improvements. In order to quantify the impact of critical sections on the overall performance, we also introduce two quantitative performance metrics of critical section bottlenecks – contention probability and size of a critical section along the critical path. Using a micro-benchmark and a set of real applications as a case study, we have shown that our method is more reliable than known methods.

We want to address the following work in our future research: critical lock analysis is not only useful in guiding the programmer for optimization opportunities but also in providing valuable guidance for optimization technologies such as accelerated critical section [25], speculative lock reordering [19] and transactional memory [12]. If one knows which locks are most critical at run time, then these technologies can achieve better performance by executing these critical locks with a higher priority.

Acknowledgment

We would like to thank Alf Larsson of Ericsson, Kun Wang, Haibo Lin, Tao Liu and Mingqiang Li of IBM Research, and the anonymous reviewers for their insightful comments that have helped enhancing the quality of the work. This research has been supported in part by a grant from Ericsson and by a grant from the Swedish Foundation for Strategic Research under the Scheme project when the first author was affiliated with Chalmers.

References

[1] P. Barford and M. Crovella. Critical Path Analysis of TCP Transactions. In *Proceedings of ACM SIGCOMM 2000*, January 2000.

[2] A. Bhattacharjee, M. Martonosi. Thread Criticality Predictors for Dynamic Performance, Power, and Resource Management in Chip Multiprocessors. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, June 2009.

[3] D. Böhme, B. R. de Supinski, M. Geimer, M. Schulz, F. Wolf: Scalable Critical-Path Based Performance Analysis. In *Proc. of the 26th IEEE International Parallel & Distributed Processing Symposium (IPDPS)*, May 2012.

[4] C. P. Breshears. Using Intel Thread Profiler for Win32 Threads: Philosophy and Theory. http://software.intel.com/en-us/articles/using-intel-thread-profiler-for-win32-threads-philosophy-and-theory/, August 2007.

[5] M. Broberg, L. Lundberg, H. Grahn. Performance Optimization Using Extended Critical Path Analysis in Multithreaded Programs on Multiprocessors. *Journal of Parallel and Distributed Computing (JPDC)*, pages 115-136, 2001.

[6] R. Bryant, J. Hawkes. Lockmeter: Highly-informative Instrumentation for Spin Locks in the Linux® Kernel. In *Proceedings of the 4th Annual Linux Showcase & Conference* - Volume 4, pages 271-282, 2000.

[7] S. Cepeda. Performance Analysis and Intel Parallel Amplifier. http://www.ddj.com/architect/217700473, May 27, 2009.

[8] J. Demme, S. Sethumadhavan. Rapid Identification of Architectural Bottlenecks via Precise Event Counting. In *Proceedings of the*

International Symposium on Computer Architecture (ISCA), June 2011.

[9] I. Dooley, L. V. Kale. Detecting and Using Critical Paths at Runtime in Message Driven Parallel Programs. In *12th Workshop on Advances in Parallel and Distributed Computing Models at IPDPS*, April 2010.

[10] S. Eyerman and L. Eeckhout. Modeling Critical Sections in Amdahl's Law and Its Implications for Multicore Design. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 362-370, 2010.

[11] B. Fields, S. Rubin, R. Bodik. Focusing Processor Policies via Critical-Path Prediction. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 74-85, 2001.

[12] M. Herlihy, J. E. B. Moss. Transactional Memory: Architecture Support for Lock-Free Data Structures. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 289-300, 1993.

[13] J.K. Hollingsworth. Critical Path Profiling of Message Passing and Shared-Memory Programs. *IEEE Transactions on Parallel and Distributed Systems*, 9(10), October 1998.

[14] K. G. Lockyer. Introduction to Critical Path Analysis. *Pitman Publishing Co.*, New York, N.Y., 1964.

[15] M. M. Michael and M. Scott. Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms. In *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 267-275, 1996.

[16] S. Olivier, J. Huan, J. Liu, J. Prins, J. Dinan, P. Sadayappan, and C.W. Tseng. UTS: An unbalanced tree search benchmark. *Lecture Notes in Computer Science*, 4382/2007:235–250, 2007.

[17] OpenLDAP Project. http://www.openldap.org

[18] A. Roy, S. Hand, T. Harris. Exploring the Limits of Disjoint Access Parallelism. In *Proceedings of HotPar '09: 1st USENIXWorkshop on Hot Topics in Parallelism (HotPar)*, pages 1-6, 2009.

[19] P. Rundberg and P. Stenström. Speculative Lock Reordering: Optimistic Out-of-Order Execution of Critical Sections. In *Proceedings of the 17th International Symposium on Parallel and Distributed Processing (IPDPS)*, pages 8-16, 2003.

[20] B. Sahelices, P. Ibáñez, V. Viñals, J. M. Llabería. A Methodology to Characterize Critical Section Bottlenecks in DSM Multiprocessors. In *Proceedings of the 15th International Euro-Par Conference on Parallel Processing (Euro-Par)*, pages 149-161, 2009.

[21] A. Saidi, N. Binkert, T. N. Mudge, S. K. Reinhardt. Full System Critical Path Analysis, In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 63-74, 2008.

[22] A. Saidi, N. Binkert, S. K. Reinhardt, T. N. Mudge. End-To-End Performance Forecasting: Finding Bottlenecks before They Happen, In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 361-370, 2009.

[23] A. Shoukry, S. Toub, P. Pennington. Identify Lock Contention Using the Visual Studio 2010 Parallel Performance Analyzer. http://code.msdn.microsoft.com/vsppalock, March 2010.

[24] SLAMD Distributed Load Generation Engine. http://www.slamd.com

[25] M. A. Suleman, O. Mutlu, M. K. Qureshi and Y.N. Patt. Accelerating Critical Section Execution with Asymmetric Multi-core Architectures. In *Proceeding of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 253-264, 2009.

[26] N. R. Tallent, J. M. Mellor-Crummey, A. Porterfield. Analyzing Lock Contention in Multithreaded Applications. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 269-280, 2010.

[27] Wikipedia. Travelling Salesman Problem. http://en.wikipedia.org/wiki/Travelling_salesman_problem

[28] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, June 1995.